2019

Augmenting American Fuzzy Lop to Increase the Speed of Bug Detection

Raviraj Mahajan
Technological University Dublin

Augmenting American Fuzzy Lop to Increase the Speed of Bug Detection

Raviraj Mahajan

A dissertation submitted in partial fulfilment of the requirements of Dublin Institute of Technology for the degree of M.Sc. in Computing (Advanced Software Development)

2019

Declaration

I certify that this dissertation which I now submit for examination for the award of MSc. in Computing (Advanced Software Development), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Dublin Institute of Technology and has not been submitted in whole or part for an award in any other Institute or University.

The work reported on in this dissertation conforms to the principles and requirements of the Institutes guidelines for ethics in research.

Signed: …….

Raviraj Mahajan

Date: 03 January 2019

Abstract

Whitebox fuzz testing is a vital part of the software testing process in the software development life cycle (SDLC). It is used for bug detection and security vulnerability checking as well. But current tools lack the ability to detect all the bugs and cover the entire code under test in a reasonable time. This study will explore some of the various whitebox fuzzing techniques and tools (AFL, SAGE, Driller, etc.) currently in use followed by a discussion of their strategies and the challenges facing them.

One of the most popular state-of-the-art fuzzers, American Fuzzy Lop (AFL) will be discussed in detail and the modifications proposed to reduce the time required by it while functioning under QEMU emulation mode will be put forth. The study found that the AFL fuzzer can be sped up by injecting an intermediary layer of code in the Tiny Code Generator (TCG) that helps in translating blocks between the two architectures being used for testing. The modified version of AFL was able to find a mean 1.6 crashes more than the basic AFL running in QEMU mode. The study will then recommend future research avenues in the form of hybrid techniques to resolve the challenges faced by the state of the art fuzzers and create an optimal fuzzing tool. The motivation behind the study is to optimize the fuzzing process in order to reduce the time taken to perform software testing and produce robust, error-free software products.

Keywords: *software testing, greybox, fuzzing, debugging, program verification, symbolic execution, software security, software validation*

Acknowledgement

I would like to thank my supervisor **Dr. Svetlana Hensman** for her expert guidance, valuable suggestions and continual encouragement and recommendations during this research project.

I would also like to take this opportunity to acknowledge all my **Dublin Institute of Technology academic staff** and especially all my professors for their help, kindness, support and knowledge.

Last and most importantly, I am eternally thankful to my parents and my friend Carla Fitzpatrick for their love, support and encouragement to pursue my masters in the area of my interest.

Table of Contents

Declaration ..... **i**
Abstract ..... **ii**
Acknowledgement ..... **iii**
Table of Contents ..... **iv**
List of Figures ..... **vi**
List of Tables ..... **vii**
List of Acronyms ..... **viii**
1. INTRODUCTION **1**
1.1 Background ***1***
1.2 Research Problem ***3***
1.2.1. Where to fuzz **3**
1.2.2. Seed Inputs or well-formed inputs **3**
1.2.3. Penetration **4**
1.2.4. Full Code Coverage **4**
1.2.5. Manual Input **4**
1.3 Research Question ***5***
1.4 Research Objective ***5***
1.5 Research Methodology ***6***
1.6 Scope and Limitation ***6***
1.7 Document Outline ***7***
2. LITERATURE REVIEW **8**
2.1 Introduction ***8***
2.2.1. Black-box fuzzing **9**
2.2.2. White-box fuzzing **10**
2.2.3. Grey-box fuzzing **10**
2.3 Discussion ***14***
3. EXPERIMENT DESIGN AND METHODOLOGY **16**
3.1 Introduction ***16***
3.2 Why American Fuzzy Lop? ***16***
3.3 How AFL works? ***17***
3.4 Instrumenting binary-only apps ***21***
3.5 Tiny Code Generator Modification ***22***

3.6 Chain Caching modification *28*
3.7 Experiment Design *29*
3.8 Conclusion *30*
4. IMPLEMENTATION, RESULTS AND ANALYSIS **31**
4.1 Introduction *31*
4.2 Implementation *31*
4.3 Experiment Results *32*
4.4 Analysis: Paired t-test *32*
4.5 Discussion *35*
5. CONCLUSION **36**
5.1 Introduction *36*
5.2 Conclusion *36*
5.2 Limitations *37*
5.3 Further Research *38*
REFERENCES **40**

List of Figures

2.2.3.1 – Validating inputs using checksum integrity checks……12

3.3.1 The UI of American Fuzzy Lop……18

3.3.2 Instrumenting software code for AFL Fuzzer……...18

3.3.3 Fuzzing with instrumented code……19

3.3.4. AFL Fuzzing with failing test case as an input……....21

4.4.1. T Distribution of the paired t-test…….34

List of Tables

4.2.1. Crashes detected in basic AFL and modified AFL……32

4.4.1. Paired t-test results for Basic AFL vs Modified AFL……34

List of Acronyms

AFL	American Fuzzy Lop
ASAN	AddressSanitizer
CGF	Coverage Based Greybox Fuzzing
GDB	GNU Debugger
IR	Intermediate Representation
JSON	JavaScript Object Notation
QEMU	Quick Emulator
SDLC	Software Development Life Cycle
SUT	Software Under Test
SAGE	Scalable, Automated, Guided Execution
TCG	Tiny Code Generator
TB	Translation Blocks

1. INTRODUCTION

1.1 Background

Software is made by people and people make mistakes (Lagus 2013). Most engineers consider "program verification research" as a typically theoretical pursuit having limited impact in the real world. Think again. Almost every piece of high-quality software has undergone rigorous tests before it is released. The Windows O/S itself adds up to a billion people affected by this field (Godefroid, Levin, & Molnar, 2012). Moreover, malware infections that prey on undiscovered bugs have increased at an alarming rate: five-fold in the past five years (Moser, Kruegel, & Kirda, 2007). It is essential to stay one step ahead.

Before the Internet became mainstream, not much attention was paid to vulnerabilities since making most of the limited resources was considered paramount (Lagus 2013, 30). According to Lagus (2013, 31), another issue with software vulnerabilities is the fact that security is often taken into consideration in the final development phases. Additionally, information systems are not simple, and they are often connected to other complex information systems (Lagus 2013, 31).

According to Bhat (2015, 23), software testing is highly complex, yet, an imperative element of any software development life cycle. Software testing should be started as early as possible (Bhat 2015, 23); however, the costs of testing are high and Godefroid et al. (2008, 30) point out that usually testing accounts for about half of the R&D budget of many software development organizations. Myers, Sandler & Badgett (2011, 5) state that in an ideal world every possible permutation of a program would be tested. However, in most cases that would not be possible or would need hundreds or thousands of possible input and output combinations and making test cases for all the combinations would be unfeasible (Myers et al. 2011, 5). Positive testing is used to confirm that the software works as it is supposed to whereas in negative testing, there is an effort to break the software (Bhat 2015, 24).

This is where fuzz testing comes into the picture. It is an integral part of the software testing process in the software development life cycle. The term fuzz testing or fuzzing is commonly used to refer to techniques which test programs through the generation of randomized input data and then running the program with those inputs (Ognawala et al., 2017). Fuzzing can be utilized to test applications where the space of conceivable sources of input is expansive. The system is utilized to perceive how well an application treats invalid inputs and, in this manner, uncover bugs. The aim of fuzzing is to traverse the maximum number of program paths and detect bugs that may present themselves as assertion violations, buffer overflows or program crashes (Pak, 2012). Better coverage of code for detection of bugs will result in more robust, error-free and secure software products.

Fuzzing has only one goal, to make the system crash (Takanen et al. 2008, 25). With fuzzing a large numbers of boundary cases are tested by either developers or quality assurance teams (Oehlert 2005, 58). The prime targets for fuzzing are input files, configuration or registry entries, APIs, user interfaces, network interfaces, database entries or command line arguments (Oehlert 2005, 59). Shapiro (2011, 58) points out that the fuzzing triggers race conditions, failures to check return code, buffer overflows and format or printf string issues.

At the structural level, fuzzing finds underflows, repetition of elements and unexpected elements (Takanen et al. 2008, 27). At the sequence level, fuzzing finds out of sequence or omitted unexpected repetition or spamming of messages (Takanen et al. 2008, 27). Fuzzing can also be a part of vulnerability analysis where fuzzing is used as a black box technique without the need of source code (Takanen et al. 2008, 102). Also whitebox fuzzing can be performed (Godefroid et al. 2012, 44).

The trade-offs between whitebox and blackbox fuzzing are different, because blackbox fuzzing is simple, easy, lightweight and fast; however, may offer only limited code coverage (Godefroid et al. 2012, 44). Whitebox fuzzing is more compound but cleverer (Godefroid et al. 2012, 44). For the discovery of bugs, Godefroid et al. (2012, 44) point out that competence of either whitebox or black box fuzzing varies according to the software under testin.

A simple blackbox fuzzing is a good start if an application has never been fuzzed and after those bugs have been found it is time to use whitebox fuzzing (Godefroid et al. 2012, 44). The effectiveness of fuzzing bases on measuring how well fuzzing covers the input space of the tested interfaces and how good the used inputs are (Takanen et al. 2008, 27–28). Fuzzers that only generate random data-based inputs are unsuccessful and find only naive coding errors (Takanen et al. 2008, 28). Godefroid et al. (2008, 32) point out that the fuzzer's ability to find errors along low probability paths are limited. Above all fuzzing is about test automation to its fullest extent (Takanen et al. 2008, 136).

1.2 Research Problem

Although fuzzing is helpful, no algorithm or technique has been able to provide a solution to comprehensively detect all bugs in the code by covering the entire code in a realistic time frame and with realistic computing power.

Fuzzing has several challenges:

1.2.1. Where to fuzz

One of the central issues to achieve deep penetration of the program is to know where to fuzz. Some parts of a binary are known to be more prone to bugs than others. It becomes important that the fuzzer be directed towards problem areas of the binary more often.

1.2.2. Seed Inputs or well-formed inputs

Another central issue is to determine the right inputs so that no time is wasted on inputs that will be rejected or lead to paths already traced. Inputs directly determine the paths and also the ability to trigger the bugs lying in those paths.

Böhme et al. (2016) states:

> Coverage-based Greybox Fuzzing (CGF) would clearly benefit from a smart seed selection if many seed files are available. (pp 1041).

This highlights the need for a smart seed generator. The challenge here is that constraint solvers in symbolic analysis do not return soon and hence achieve incomplete code coverage under limited time (Ognawala, Hutzelmann, Psallida, & Pretschner, 2017).

1.2.3. Penetration

The fuzzer should be able to traverse deep into the program. Otherwise it will only be able to detect surface bugs as is the case in most blackbox fuzzers (Godefroid et al., 2012). Path explosion as discussed earlier is the main challenge to deep penetration.

1.2.4. Full Code Coverage

Inputs and fuzzer logic should complement each other in a way not only to traverse deep but also to be able to cover maximum code possible. It is possible there might be severe bugs in a component of the program that was expected to be bug-free.

1.2.5. Manual Input

The best well-formed inputs are still generated manually. Manual input can be error prone if the programmer is inexperienced. Although, AFL was able to recreate meaningful input files from an empty file, that does not translate to effective input seeds (Böck, 2015). Creating tests manually is expensive, error-prone and most of the time inconclusive (Burnim, 2008).

However, the most important constraint when it comes to software testing is time. Given unlimited time, any software can be tested to be completely bug free. But this is not the case. The immense complexity of the latest software products would require an unfeasible amount of time for all execution paths to be fuzzed to exhaustion. There is a need to speed up the fuzzing process.

1.3 Research Question

The research conducted aims to investigate the following research question:

Can implementing an intermediate layer in the tiny code generator of the American Fuzzy Lop fuzzer improve its speed significantly?

To elaborate, this study will assess whether the speed of the fuzzing process under emulation mode of the AFL fuzzer can be improved significantly by introducing an intermediate layer in the tiny code generator to help the translation of transfer blocks between the source and the target architectures.

1.4 Research Objective

This research intents to investigate the state-of-the-art fuzzing techniques and any improvement that can be made. This will be done be conducting an experiment by testing benchmark binaries with known bugs. The AFL fuzzer will be modified and compared against the baseline AFL fuzzer. The objective is to improve the fuzzer's performance under the QEMU (Quick Emulator) emulation mode. The study will also provide the results and conclusion for additional research and studies in the future. It is expected that this study will add to the knowledge in the information security area specially the field of software testing and lead to more safe and robust software products. The research objectives are:

- To assess related studies which have made contributions in the field of fuzzing. Various studies in the past and their findings will be collected and studied carefully.
- To test the efficiency of current state of the art fuzzers like AFL.
- To test if an intermediate layer in the tiny code generator to help the translation of transfer blocks between the source and the target architectures leads to faster performance or not.
- To test if the modified fuzzer reveals previously unknown bugs.

- To recommend further avenues of research that could help improving fuzzing techniques and their performance.

1.5 Research Methodology

The main aim of this research is to determine whether the modified AFL fuzzer can detect bugs faster than the normal AFL under QEMU mode. This will require a quantitative analysis to be undertaken.

To accomplish this, an experiment will be conducted to test the performances of the individual fuzzers. The test subjects will be ten different binaries which are commonly used as benchmarks for testing fuzzing techniques. Each binary will be tested for one hour with both the fuzzers. The results will then be compared using a paired t-test evaluation to check if any significant difference in performance was achieved.

1.6 Scope and Limitation

Fuzz testing is an integral part of the software testing process in the software development life cycle. The term fuzz testing or fuzzing is commonly used to refer to techniques which test programs through the generation of randomized input data and then running the program with those inputs (Ognawala et al., 2017). The aim of fuzzing is to traverse the maximum number of program paths and detect bugs that may present themselves as assertion violations, buffer overflows or program crashes (Pak, 2012). Better coverage of code for detection of bugs will result in more robust, error-free and secure software products.

The limitation here is that only QEMU mode of the AFL fuzzer is being modified, the performance under normal mode will not be affected. Also, only ten binaries are being tested each for only one hour due to time and resource limitations.

1.7 Document Outline

Chapter 2; Literature Review: This chapter explores fuzzing and its role in the SDLC. It further discusses the state-of-the-art fuzzers and their strengths and limitations and challenges in the field of automated software testing

Chapter 3; Experiment Design and Methodology: This chapter explains the working of AFL fuzzer and outlines the modifications made in an attempt to improve its performance. It delineates the experiment to take place.

Chapter 4; Implementation, Results and Analysis: This chapter discusses the precise steps in the experiment. It contains the results and their analysis using a paired t-test.

Chapter 5; Conclusion: This chapter expounds on the results and delineates the limitations of the experiment conducted. It also mentions the future research ideas in order to further enhance the fuzzing process.

2. LITERATURE REVIEW

The chapter provides a brief but detailed review of relevant literature concerning the current approaches to detecting vulnerabilities. An overview of fuzzing including the need for fuzzing, different types of fuzzers and their strategies are included. The state of the art fuzzers and their advantages and disadvantages are discussed in this chapter.

2.1 Introduction

Modern methods for discovering programming vulnerabilities can be partitioned right off the bat into two distinct methodologies; static analysis and dynamic analysis. Static analysis of programming includes methods for looking at the source code or a binary that is compiled without executing it. Dynamic investigation includes inspecting the software at runtime, commonly subsequent to connecting some sort of debugger to it. Both approaches have their relative advantages and disadvantages (Shoshitaishvili, 2016).

For static analysis, several automated tools exist that can be combined with manual code review by a skilled analyst. The less sophisticated tools essentially just scan the target source code looking for known dangerous functions such as strcpy() in C programs. The more advanced tools often work using some sort of taint analysis (Dahse, 2014). These tools will distinguish and 'taint' any variable that has its value set from info that enters the objective application from a client. This tainted input and its impact on other information will at that point be followed as it passes through the source code. At whatever point it is seen that corrupted information could come to a 'sink' or conceivably unsafe function this will be hailed for further examination. Static analysis, although useful, often produces many false positives that cannot be exploited in practice and requires a lot of manual verification work to identify which issues are genuine vulnerabilities (Dahse, 2014). It does, however, allow for complete code coverage with the entire application being inspected.

In the case of dynamic analysis, the most common automated method for discovery of vulnerabilities is the method of fuzz testing or fuzzing. Essentially this consists of repeatedly giving an application invalid input and monitoring for any sign of this triggering a bug, such as the application crashing or hanging (Gadi, 2007). There are numerous advantages to this methodology, for example, the simplicity of automation and the capacity to test even extensive applications where code survey would be too tedious. Moreover, each bug found by fuzzing naturally accompanies its own 'proof of concept' experiment demonstrating that the bug can be activated by a client. Fuzzing is the primary method used nowadays to detect high profile vulnerabilities.

The motivation behind fuzzing and any research pertaining to techniques for finding vulnerabilities, can be offensive or defensive. Software organizations utilize these procedures to recognize vulnerabilities in their own products and is considered an integral part of the Software Development Life Cycle (SDLC). Whitehat security specialists likewise utilize these strategies to discover vulnerabilities in both open source and restrictive software and inform the software developers about the vulnerabilities to allow them to be able to release a patch.

But various intelligence agencies, defense contractors, and even organized criminals are known to use these techniques for malicious purposes. These organizations benefit from keeping the discovered vulnerabilities to themselves and even exploiting them if need be.

2.2. Types of fuzzing:

Fuzzing is classified into two main types as discussed below:

2.2.1. Black-box fuzzing

Traditional fuzzing is called black box fuzzing. This is the simplest form of fuzzing and assumes that the input as well as output of the SUT (System Under Test) are the only things the fuzzer knows. The internal working of the SUT is

not known, hence making it a black-box. For instance, in a network protocol, both the server-side and client-side code could be fuzzed for vulnerabilities (Takanen, Demott, & Miller, 2008). An aggressor could leave the fuzzing procedure running until the point that a bug is uncovered. Since protocol implementations may be the same on numerous servers, an assailant could set up a similar framework to run the fuzzing procedure against. In the event that a bug is found, it can in principle, be misused on each server running a similar implementation of that protocol.

2.2.2. White-box fuzzing

White-box fuzzing, on the other hand, utilizes program analysis to know the effect of the input and increment code coverage of the SUT. White-box fuzzing exploits its access to the source code and design particulars of the SUT. Symbolic execution is very connected to white-box fuzzing and is a method for deciding how inputs propagate different paths when the program executes. When symbolic execution is performed, input variables are assigned symbolic values rather than concrete values. All changes to the symbolic value are stored and taken into account later when an if statement is reached. This empowers the symbolic execution to set the symbolic values such as to take a specific path according to the changes to the symbolic value. While symbolic execution takes place, constraints can be stored. The constraints are assembled from conditional paths of execution experienced along the execution, invalidated, and solved utilizing a constraint solver. The output of the solver is then used to create new input variables. These input variables are then used to find new paths or uncover security risks or bugs. In the development of Windows 7, white-box fuzzing was primarily implemented and found one-third of the total vulnerabilities found prior to the release (Bounimova, Godefroid, & Molnar, 2013).

2.2.3. Grey-box fuzzing

The term grey-box fuzzing coined in 2007 by Demott, Enbody and Punch. Despite being bound to only provide input and only be able to look at the output

of the SUT, black-box fuzzers can still be smart in the way they generate inputs, and therefore achieve high code coverage without suffering from problems in scalability (Kargen & Shahmehri, 2015) (Rawat et al., 2017). Although black-box fuzzing can achieve high code coverage, the lack of knowledge of the working of the SUT makes it difficult to execute certain paths. White-box fuzzing is able to solve this issue using symbolic execution. Grey-box fuzzers are known to use symbolic execution as well as implementing dynamic taint analysis. They combine the best of both worlds and most state of the art fuzzers are considered to be of the grey-box variety.

Some of the state of the art fuzzers are discussed below:

A. SAGE – the first symbolic execution based whitebox fuzzer

Prior to 2008, blackbox fuzzing was the norm for software testing. It is a form of blackbox random testing where randomly mutated well-formed inputs are run on the program under test. In some cases, grammars are implemented to create the inputs based on application-specific knowledge (Bounimova, Godefroid, & Molnar, 2013). Although blackbox fuzzing was effective, it provided very low code coverage. Consider the following conditional statement as put forth by Bounimova et al. (2013):

```
                int foo(int x) {
                                                                /
/ x is an input
                int y = x + 3;
                if (y == 13) abort(); // error
                return 0;
                }
```

This only has a 1 in 2^{32} chance of being accessed since the input variable x would have a randomly-allocated 32-bit value. This low code coverage meant many bugs were not detected.

In 2008, Patrice Godefroid, Michael Y. Levin and David Molnar implemented symbolic execution and dynamic test generation to give rise to the whitebox fuzzing tool, SAGE (Scalable, Automated, Guided Execution). This algorithm could dynamically create randomly mutated input variables based on constraints in the program under test. This way the well-formed inputs were able to access more paths in the program allowing for more code coverage.

SAGE was credited with finding roughly one third of all the bugs during the development of Microsoft's Windows 7. However, majority of the test time was spent in the discovery of the appropriate inputs which was done manually by a skilled programmer. SAGE set the stage for further research to overcome its drawbacks.

B. TaintScope

TaintScope, built on the symbolic execution engine mentioned in SAGE, is a fuzzer that "can symbolically evaluate a trace, reason about all possible values that can execute the trace, and then detect potential vulnerabilities on the trace" (Wang, Wei, Gu, & Zou, 2010).

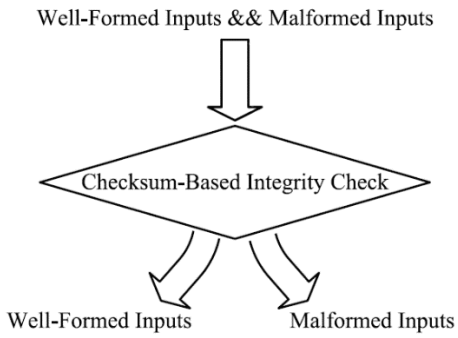

Figure 2.2.3.1 – Validating inputs using checksum integrity checks (Wang et al., 2010).

As observed in figure 1, it uses checksum verification to verify the integrity of the mutated well-formed inputs, saving a lot of time by getting rid of malformed inputs. It also intelligently focuses on modifying those bits of the input that can mostly result in triggering exceptions in the program.

TaintScope was able to find 27 bugs that were previously unknown in many widely used applications, including Microsoft Paint, Google Picasa, ImageMagick, and Adobe Acrobat.

C. American Fuzzy Lop (AFL)

AFL is a well-known open source, off-the-shelf fuzzer first developed in 2014 and has been updated regularly making it a benchmark for most novel algorithms being researched. Some of the enhancements made to AFL are credited to scientists incorporating their fuzzing techniques into AFL (Stephens et al.,2016) (Bohme, 2016), which is a strong indication that AFL can be considered one of, or even, the foremost grey-box fuzzer currently.

It emphasizes unique code coverage i.e. creating inputs that trace paths that have not been accessed before (Zalewski, 2014). Another noteworthy feature of AFL is that it keeps a record of all the loops that it gets into and decides in the end which would be most important path to trace for a particular loop, thereby reducing the time complexity from N^2 to log(N) (Stephens et al., 2016).

A demonstration of AFL entailed using a bogus file as input to fuzz the djpeg tool that comes with libjpeg. As time passed, the fuzzer automatically created a new input file with a valid JPEG header (Böck, 2015). This means that smart fuzzers can operate without any seed input.

D. Driller

Driller, a fuzzing tool developed at UC Santa Barbara last year, builds on the AFL. It uses the complimenting strengths of fuzzing and symbolic execution to allay their respective drawbacks. In SAGE, symbolic execution led to the path explosion problem where too many possible paths are generated. A trade-off had to be made between time and code coverage (Bounimova et al., 2013). Also, a lot of time was spent on manually creating inputs.

Driller solves these issues by using concolic execution to split the program into compartments and then use fuzzing to drill deep into the program by using

valid inputs crafted by the concolic execution engine (Stephens et al., 2016).

Tests were performed on 126 binaries using AFL and Driller. AFL found 66 bugs while Driller was able to find 72 bugs (Stephens et al., 2016).

E. AFLFast

AFLFast is a state of the art fuzzing tool that, as the name suggests, is an augmentation on AFL that implements Coverage-based Greybox Fuzzing (CGF). CGF combines whitebox and blackbox fuzzing as it doesn't need any program analysis to generate seed inputs. Instead of symbolic execution, it uses Markov chains to recognize paths that are not being frequently traced by the fuzzer. This improves code coverage without requiring additional time for input generation.

It was able to find more bugs in lesser time as compared to off-the-shelf AFL (Böhme, Pham, & Roychoudhury, 2016).

F. Neural Byte Sieve

Rajpal, Blum and Singh (2017) developed a technique to use neural networks to discover and learn patterns in the input files by studying previous fuzzing runs. These patterns can then be leveraged to predict optimal locations to perform future fuzzing using the right mutations.

They implemented this in AFL and were able to demonstrate significant enhancements in terms of code coverage and unique code path traces.

2.3 Discussion

Ever since SAGE (Godefroid et al.,2012) there has been a race to develop better automated seed inputs or better fuzzer logic to overcome the challenges. The primary strength of grey-box fuzzing is that it functions without requiring the source code of the SUT. This is advantageous for the testing and security verification of third-party softwares (DeMott, 2007). Since grey-box fuzzing is

based on the lightweight black-box fuzzing technique, but may still glean information about the SUT, code coverage can still be leveraged without sacrificing time on program analysis (Bayer et al., 2006). Grey-box fuzzing, however, is unsuitable for conditions when the internals of the software cannot be accessed.

The state of the art is to combine lightweight blackbox/greybox fuzzers with whitebox fuzzers that employ symbolic execution to combine the strengths of both and minimize each other's drawbacks. They build a program model automatically as it is tested, allowing for optimal guidance of the fuzzer. For example, HybridFuzz (Pak, 2012) first runs symbolic execution to create inputs that lead to "frontier nodes" which are then used by a blackbox fuzzer. In contrast, Driller starts with AFL and turns to symbolic execution when it "gets stuck", for example, to generate a magic number (Stephens et al., 2016). Whereas AFLFast uses Markov Chains for the same (Böhme et al., 2016).

Time is an important constraint when it comes to software testing. Grey-box fuzzers are more non-specific than white-box fuzzers. Since white-box fuzzers take into consideration the source code, information about the language's syntax is essential to perform the analysis. Hence it can be difficult to test software's that use multiple languages or those whose language is unknown to the fuzzer. Grey-box fuzzers unlike black-box fuzzers can automatically learn to generate valid inputs which eliminates the need to have seed inputs. By not being dependent on valid seed inputs that have the purpose of taking the execution away from branches that only lead to exception handling in the parser code, grey-box fuzzers can be automated in a more sophisticated matter than black-box fuzzers.

3. EXPERIMENT DESIGN AND METHODOLOGY

3.1 Introduction

In the last chapter, the challenges faced by fuzzing were discussed. Time being one of the primary ones. In this chapter, the basic working of American Fuzzy Lop (AFL) fuzzer will be discussed followed by the modifications proposed and their effect on the fuzzing process. A quantitative experiment will be discussed in this chapter that compares the basic AFL to the modified version which is expected to improve the runtime and hence speed up bug detection.

3.2 Why American Fuzzy Lop?

An effective fuzzing framework comprises of more than just a method to create and pass invalid inputs to the software being tested. Some method for instrumenting the software is required so as to screen what is happening inside as each case is handled. Exceptions, crashes and other unexpected behavior must be identified and additionally logged and answered to the client alongside as much information as would be essential for further manual examination. There must likewise be a test harness that can restart the application automatically as required so that the fuzzing effort can run unsupervised.

In Franz (2010) and Vimpari (2015), several prevalent free fuzzing tools are assessed and compared to each other. This work outlines well that there are a wide range of sorts of fuzzers and they are appropriate for various circumstances. There is no single tool that gives the best results in each circumstance.

For the experiment, an open source state of the art fuzzer called the American Fuzzy Lop as discussed in last chapter will be implemented.

American Fuzzy Lop Fuzzer or AFL is a well-known open source, off-the-shelf fuzzer first developed in 2014 and has been updated regularly making it a benchmark for most novel algorithms being researched. Some of the enhancements made to AFL are credited to scientists incorporating their fuzzing techniques into AFL (Stephens et al.,2016) (Bohme, 2016), which is a strong

indication that AFL can be considered one of, or even, the foremost grey-box fuzzer currently. Hence, AFL was chosen to be augmented.

3.3 How AFL works?

AFL Fuzzer is a security focused brute-force fuzzer that can be used either in compile time instrumentation mode or in traditional blind fuzzer mode (Zalewski, 2016). Instrumentation can be either done either while compiling or by the use of QEMU hypervisor (Stephens et al. 2016). American Fuzzy Lop employs a altered form of edge coverage in order to pick up small, local-scale changes to program control flow (Zalewski, 2016). Input generation is done by a genetic algorithm, mutating inputs based on the genetics inspired rules and ranking them by a fitness function (Stephens et al. 2016). Fitness functions base on unique code coverage where an execution path is triggered, which is different from the paths triggered by other inputs (Stephens et al. 2016). Union of control flow transitions, which American Fuzzy Lop has seen from its inputs, such as tuples of the source and destination basic blocks are tracked by American Fuzzy Lop (Stephens et al. 2016).

The inputs that make an application execute in a different way get prioritized in the generation of future inputs (Stephens et al., 2016). In order to reduce the size of the path spaces for loops, American Fuzzy Lop uses a heuristic approach where only log(N) paths are taken into account for each loop instead of N paths (Stephens et al., 2016). Randomization of the programs interferes with the genetic fuzzer's evaluation of inputs because an input, which produces interesting paths under a certain random seed may not do so under another random seed (Stephens et al., 2016). If randomization is not removed, the fuzzing component is likely to explore only few paths, but if constant randomness is used, then the program accepts the same input each time and that allows the fuzzer to find this value and subsequently explore further (Stephens et al., 2016).

The process of American Fuzzy Lop is to load user-supplied initial test cases into the queue, then take next input file from the queue, attempt to trim the test case to the smallest size, which will not change the measured behavior of the system under

test, mutate the file repeatedly by using a variety of traditional fuzzing strategies and if any of the generated mutations caused new state transitions that were recorded by the instrumentation, new entry of the mutated output is added to the queue and then the algorithm takes next input file from the queue and repeats (Zalewski, 2016).

Figure 3.3.1 The UI of AFL

If instrumentation is to be used, the fuzzed program has to be instrumented with aflgcc (Zalewski, 2016). The instrumentation will also display the number of locations that were instrumented, as seen in Figure 3.3.2.

Figure 3.3.2 UI showing instrumented locations

During instrumentation assembly code is injected to the target program that is used to trace executions paths as new inputs are entered (Margaritelli, 2015). Injected assembly code is also used to determine if known or unknown execution paths is triggered by a new mutation input (Margaritelli, 2015). When fuzzing with

instrumented code, last new path should show the time when last new path was found like in Figure 3.3.3 (Zalewski, 2016).

Figure 3.3.3 Running AFL withs instrumented code.

The disparity between instrumented and non-instrumented afl-fuzzing can be seen using a simple program that is known to have only one unique crash. Non-instrumented fuzzing shows total paths as 1 and total crashes to be 0. On the other hand, performing fuzzing with instrumentation, the total paths were found to be 5 and total crashes were 63 with only 1 unique.

American Fuzzy Lop fuzzes until Ctrl-C is pressed but at least one queue cycle should be completed before fuzzing is stopped (Zalewski, 2016). Completing one queue cycle may take from seconds to even a week (Zalewski, 2016). The fuzzing is performed by afl-fuzz utility that requires a read-only directory with initial test cases, a directory to store results and path to the binary to be fuzzed (Zalewski, 2016).

For example, when command

```
./afl-fuzz -i input1 -o output1 /home/virtual/Ravi/afl-1.92b/a.out
```

is used, the -i parameter points out to a directory with initial test cases and -o parameter points out to a directory to store the fuzzing results (Zalewski, 2016).

American Fuzzy Lop comes with sample test cases containing small standalone files that can be used to seed afl-fuzz (Zalewski, 2016). The archives directory has, among others, samples of rar, tar and zip (Zalewski, 2016). Images directory has, among others, samples of bmp, jpeg and png (Zalewski, 2016). Multimedia directory has a sample of h264 and others directory has among others samples of js, pdf, rtf and text files (Zalewski, 2016). These will be used as benchmarks for the experiment.

The directory to store results will have three subdirectories that are updated in real time (Zalewski, 2016). Queue directory has test cases for every distinctive execution paths and the starting files given by the user (Zalewski, 2016). Crashes directory has unique test cases that caused the program to receive a fatal signal and the entries are grouped by the received signal (Zalewski, 2016). Hangs directory has unique test cases that cause the tested program to time out (Zalewski, 2016).

American Fuzzy Lop considers crashes and hangs unique if the associated execution paths involve any state transitions that have not been seen in previously recorded faults (Zalewski, 2016). Crash is considered unique if the crash trace includes not seen a tuple in any of the previous crashes or if the crash trace is missing a tuple that was present every time in earlier faults (Zalewski, 2016).

In order to ease crash analysis American Fuzzy Lop fuzzer has a crash exploration mode where a crashed test case is provided as an input and American Fuzzy Lop uses its genetic algorithms to see how far can be reached within the instrumented codebase while the program is kept in the crashing state (Zalewski, 2016). Figure 3.3.4 shows an example where the crashes are being used as inputs, though, only one unique crash is recognized by the fuzzer.

Figure 3.3.4. Fuzzing with a failing test code as input.

American Fuzzy Lop produces a coverage-based grouping of crashes that can be triaged manually or use GDB scripts to analyze (Zalewski, 2016). Also, every crash can be traced to its parent non-crashing test case in the queue, which should make it easier to detect faults (Zalewski, 2016). Zalewski (2016) points out that some crashes produced by fuzzing can be fairly difficult to evaluate for exploitability without substantial work in debugging and code analysis.

3.4 Instrumenting binary-only apps

When the source code is unavailable, AFL offers some experimental support for fast, convenient instrumentation of binaries that are black-box. This is carried out with a form of QEMU running in the "user space emulation" mode. According to Zalewski (2016), this mode is about 2-5x slower compared to compile-time instrumentation, is less prone to parallelization, and may present some other quirks as well.

Full system testing is difficult to achieve because most testing tools do not have full access to the lower levels of the software stack. To address this issue, virtualization technology is frequently used in full system testing tools. One such virtualization technology is QEMU (Quick Emulator). It is an open source emulator software that carries out hardware simulation.

QEMU's aim is to emulate a target on top of a host, each having different or similar architectures. A simple technique would be to code an interpreter for the target's instruction set followed by compiling it on the host system. However, this is a very time-consuming endeavour. A smarter method is just-in-time compilation: interpret the target's code to the native host instructions and then execute at native speed which is accomplished by QEMU automatically.

3.5 Tiny Code Generator Modification

This raises another issue, translating directly from target to host is not able to scale well, as it would require translators for all the target and host tuples. This can be solved by introducing an indirection layer, Tiny Code Generator (TCG).

A TCG front-end takes native target instructions and puts them into an architecture-independent intermediate representation (IR). A TCG back-end then casts the IR into the native host's instructions. A new targer architecture then only requires the programmer to write a new front end while a new host architecture would require only a new backend. This immensely lowers the manual work required to carry out the fuzzing.

The translation is done while running during emulation at the elementary block level. Since translation is resource intensive, translation blocks (TBs) are kept in the TCG cache, where from they can be called if they are executed again.

An important issue to take into consideration here is that the memory layout of the code that was translated would not always be a match for the original code. References to memory addresses also need to be fixed.

For example, the control-flow instruction that would terminate a block. If it is a direct jump then the address of the destination is already known, so it can be

directly corrected, and the jump can then be translated into a native jump onto the inheritor, resulting in a zero runtime overhead. QEMU calls this phenomenon block chaining. In the case of an indirect jump, however, the destination cannot be determined at the time of translation. In such a case, the jump can be translated to a call back to the QEMU's core, which will then consequently translate the destination block and transfer control to it, thus continuing the emulation. Clearly, this will have a performance price.

AFL, being a coverage-guided fuzzer, needs a tracing instrumentation to collect information about the program's control flow (Biondo, 2018). If you have the program's source code, you can recompile it using AFL's instrumenting compiler, which will add a small snippet to the beginning of every basic block. When you only have a binary, you can use AFL's QEMU mode: the binary runs within a patched QEMU that collects coverage information and delivers it to AFL (Biondo, 2018).

QEMU patches in AFL function as follows:

The qemu_mode/patches/afl-qemu-cpu-inl.h file contains the actual implementation. This has two main components: the forkserver and the tracing instrumentation (Biondo, 2018). The forkserver is AFL's method to optimize the initialization overhead as discussed. The forkserver starts before the program is run and hence the children always have a vacant TCG cache (Biondo, 2018). Therefore, there's a technique by which children notify the parent of the newly translated blocks, leading the parent to translate the block within its own cache for upcoming children.

The instrumentation part then calls the accel/tcg/cpu-exec.c in the QEMU core. Specifically, this patch inserts a snippet of code into cpu_tb_exec, which is called whenever a TB is executed by the emulator. The patch calls afl_maybe_log, which holds the responsibility to check whether the block resides within the traced bounds and, if so, the control flow transfer is traced into AFL's edge map.

This gives rise to an issue, the jumps in the chained blocks will be unable to call back into the emulator and therefore, it won't go through cpu_tb_exec. AFL provides a s[i]olution to this is disabling chaining:

setenv("QEMU_LOG", "nochain", 1);

However, this causes it to function very slowly since direct jumps are not being traced at all.

TCG Instrumentation:

The idea is to handover the instrumentation into the translated code by adding a small snippet of TCG IR at the start of every TB. By doing this, the instrumentation becomes a part of the emulated software, thus eliminating the need to go back into the emulator at each block allowing us to re-enable chaining in order to reduce the time required.

The function afl_maybe_log exists in the file qemu_mode/patches/afl-qemu-cpu-inl.h:

```
/* This is the same as the tuple logging function from afl-as.h. */

static inline void afl_maybe_log(abi_ulong cur_loc) {

  static __thread abi_ulong prev_loc;

  /* Optimize for cur_loc > afl_end_code, only for Linux. */

  if (cur_loc > afl_end_code || cur_loc < afl_start_code || !afl_area_ptr)

    return;

  /* It is seen that QEMU always maps to some fixed locations, so ASAN will not be a concern. But the instruction addresses may be aligned inadvertently. The value has to be randomized to get something quasi-uniform. */

  cur_loc  = (cur_loc >> 4) ^ (cur_loc << 8);

  cur_loc &= MAP_SIZE - 1;
```

```
  /* Here probabilistic instrumentation is implemented by
reading the scrambled block address. This will keep the
instrumented memory locations steady across various runs. */

  if (cur_loc >= afl_inst_rms) return;

  afl_area_ptr[cur_loc ^ prev_loc]++;

  prev_loc = cur_loc >> 1;

}
```

Everything that depends on cur_loc can be done at translation time, as cur_loc is the address of the current block. Basically, TCG IR needs to be generated for the last two lines. This can be done as follows:

```
/* Generates TCG code for AFL's tracing instrumentation. */

static void afl_gen_trace(target_ulong cur_loc)

{

 static __thread target_ulong prev_loc;

 TCGv index, count, new_prev_loc;

 TCGv_ptr prev_loc_ptr, count_ptr;


 /* Optimize for cur_loc > afl_end_code, which is the case on Linux systems. */


 if (cur_loc > afl_end_code || cur_loc < afl_start_code || !afl_area_ptr)

  return;
```

```
/* QEMU always maps to fixed locations, so ASAN is not a concern. But instruction addresses may be similar. The value needs to be changed to get a quasi-uniform value. */

cur_loc  = (cur_loc >> 4) ^ (cur_loc << 8);
cur_loc &= MAP_SIZE - 1;

/*  Instrumentation is implemented by observing the scrambled block address as this stabilizes the locations across the various runs. */

if (cur_loc >= afl_inst_rms) return;

/* index = prev_loc ^ cur_loc */
prev_loc_ptr = tcg_const_ptr(&prev_loc);
index = tcg_temp_new();
tcg_gen_ld_tl(index, prev_loc_ptr, 0);
tcg_gen_xori_tl(index, index, cur_loc);

/* afl_area_ptr[index]++ */
count_ptr = tcg_const_ptr(afl_area_ptr);
tcg_gen_add_ptr(count_ptr, count_ptr, TCGV_NAT_TO_PTR(index));
count = tcg_temp_new();
tcg_gen_ld8u_tl(count, count_ptr, 0);
tcg_gen_addi_tl(count, count, 1);
tcg_gen_st8_tl(count, count_ptr, 0);
```

```
  /* prev_loc = cur_loc >> 1 */
  new_prev_loc = tcg_const_tl(cur_loc >> 1);
  tcg_gen_st_tl(new_prev_loc, prev_loc_ptr, 0);
}
```

This should be called before translating each block. The TB IR generation happens in tb_gen_code (accel/tcg/translate-all.c), which subsequently calls the target frontend's gen_intermediate_code function:

```
tcg_ctx.cpu = ENV_GET_CPU(env);
gen_intermediate_code(cpu, tb);
tcg_ctx.cpu = NULL;
```

This needs to be changed to insert the IR before each block:

```
tcg_ctx.cpu = ENV_GET_CPU(env);
afl_gen_trace(pc);
gen_intermediate_code(cpu, tb);
tcg_ctx.cpu = NULL;
```

The function setenv("QEMU_LOG", "nochain", 1) has to be removed from the 4 AFL files: afl-analyze.c, , afl-tmin.c, afl-fuzz.c and afl-showmap.c.

3.6 Chain Caching modification

AFL uses a forkserver strategy to decrease and optimize initialization overhead. Basically, the forkserver is run after initialization, and it forks off children at AFL's request. Each child then executes a test case. This approach removes QEMU's initialization overhead but can cause TCG cache thrashing since the parent, after being initialized, has a vacant TCG cache, thus all the children would be starting with an empty cache.

To circumvent this, AFL's patches establish a connection between the parent and child, which the child uses to alert the parent of each new basic block translation that takes place. The parent will then be able to translate the block within its own cache thus making it accessible to future children.

To do this, AFL patches tb_find in accel/tcg/cpu-exec.c by putting a call to afl_request_tslafter tb_gen_code, which does the translation of the block. The afl_request_tsl function sends the data needed to identify the TB (address, CS base and flags) to the parent, that is spinning in afl_wait_tsl. Finally, afl_wait_tsl calls the tb_gen_code function to translate the block residing in the parent's cache.

The tb_find function then receives two parameters, last_tb and tb_exit, which identify the previous TB and slot where it jumps of the previous TB's last instruction, which led to the current one. After translating the recieved block, tb_find performs the chaining by patching the previous block's jump slot:

```
/* Patching the calling TB. */
if (last_tb && !qemu_loglevel_mask(CPU_LOG_TB_NOCHAIN)) {
    if (!have_tb_lock) {
        tb_lock();
        have_tb_lock = true;
    }
    if (!tb->invalid) {
        tb_add_jump(last_tb, tb_exit, tb);
```

```
    }
}
```

However, afl_wait_tsl won't cache the chains between TBs. The implementation for caching of the patched jump slots can be found in the appendix. Basically, the parent is notified when the tb_add_jump block is reached so the caching can take place.

3.7 Experiment Design

The summary statistics produced by the AFL include the number of bugs found, code coverage percentage and time taken among several other relevant parameters. A comparison can be made between the number of bugs for the two fuzzer configurations tested. This relates directly to the research question. There is expected to be a significant rise in both these numbers in the modified AFL fuzzer compared to the basic AFL fuzzer.

The experiment consists of running both versions of the fuzzer on ten common benchmark binaries for one hour per binary. The number of bugs found will be counted. If the number of bugs found by one version is higher than the other, it can be inferred that that particular version of the fuzzer is performing faster than the other.

Several benchmark binaries are provided in the AFL package. The ones chosen for the experiment are as follows:

1. tcpdump
2. Readelf
3. nm
4. objdump
5. C++filt
6. xmllint
7. mutooldraw
8. djpeg
9. readpng
10. strings

Statistical analysis can be done for the two configurations of the fuzzer to find the mean difference in performance using a paired t-test. If improved performance is observed overall in the ten binaries tested, then the hypothesis can be said to have been proven unequivocally.

3.8 Conclusion

Time is of the essence. This is true for fuzzing as well. In today's world with rapid software development and quick updates, it is essential that software testing can be done as fast as possible. In this chapter, the AFL fuzzer's inner workings were discussed along with the modifications proposed to reduce the time taken to find crashes. The experiment design was explained as well. The next step would be to perform the experiment and record the results.

4. IMPLEMENTATION, RESULTS AND ANALYSIS

4.1 Introduction

As discussed in the previous chapter, the basic AFL fuzzer and the modified version will both be run on ten different binaries and the number of bugs detected, code coverage achieved, and time taken for each will be noted. Due to time constraints, each binary will be tested only for one hour using both the fuzzers.

The results will be compared using a paired t-test as detailed in the section 4.3 below.

4.2 Implementation

The programming language used will be C in a Linux Environment on an x86 machine with 4 cores.

The experiment comprises of the following basic steps:

1. First, prime the AFL by initializing it in the QEMU mode.
2. We run the fuzzer on the ten benchmark binaries selected.
3. After one hour has passed, the fuzzer's execution is stopped and the number of bugs detected for each of the benchmark binaries are recorded.
4. Then, the modified version of AFL is initialized in QEMU mode.
5. Steps 2 and 3 are repeated for this version of the fuzzer.
6. The observations are tabulated, and a paired t-test is conducted to determine whether any significant change has been achieved in the number of bugs detected.

4.3 Experiment Results

Both fuzzers were run for one hour each on the ten binaries and the number of crashes detected was noted down. The following table shows the values observed.

Binary Under Test	Crashes detected using basic AFL	Crashed detected using modified AFL
tcpdump	12	15
Readelf	19	21
nm	6	6
objdump	8	11
C++filt	14	16
xmllint	32	37
mutooldraw	9	11
djpeg	8	6
readpng	17	19
strings	6	5

Table 4.2.1. Crashes detected in basic AFL and modified AFL

The fuzzing benchmarks are derived from real-life libraries that have a wide range of bugs and hard-to-find code paths for bug finding tools and are included in the AFL package.

Some examples include jpeg, png, xml, JSON, SSL, etc. The bugs present in these binaries are well-documented and hence the fuzzers can be tested to find how many of the known bugs were discovered by it and also the different paths explored by the fuzzer which can tell us the code coverage achieved.

4.4 Analysis: Paired t-test

This section delineates the technique that will be utilized to dissect the analysis results. A paired t-test method will be implemented in this analysis. It is a factual

system that is utilized in "before-after‟ studies, case-control study, or coordinated sets (Sigma, 2009). It analyses two means that are obtained from a similar subject. The point of such sort of test is to decide if the mean difference between the observations on an explicit result is significantly different from zero.

In this investigation, the data contains two factors that are tested on the same set of variables. The two factors are the basic AFL and the modified AFL and the variables are the number of bugs found in the ten binaries under test. By utilizing the paired t-test, it becomes clear whether the modifications made to AFL were successful in speeding up the fuzzing or not.

Hypothesis:

The 'null hypothesis' will be as following:

H0: There is no significant difference in mean of the number of bugs found by the basic AFL and the modified AFL

And the 'alternative hypothesis' can be defined as:

H1: There is a significant difference in mean of the number of bugs found by the basic AFL and the modified AFL

The results of the paired t-test are as follows:

P value and statistical significance:

The two-tailed P value equals 0.0368

Confidence interval:

The difference in the mean of Modified AFL and Basic AFL minus was found to be 1.60

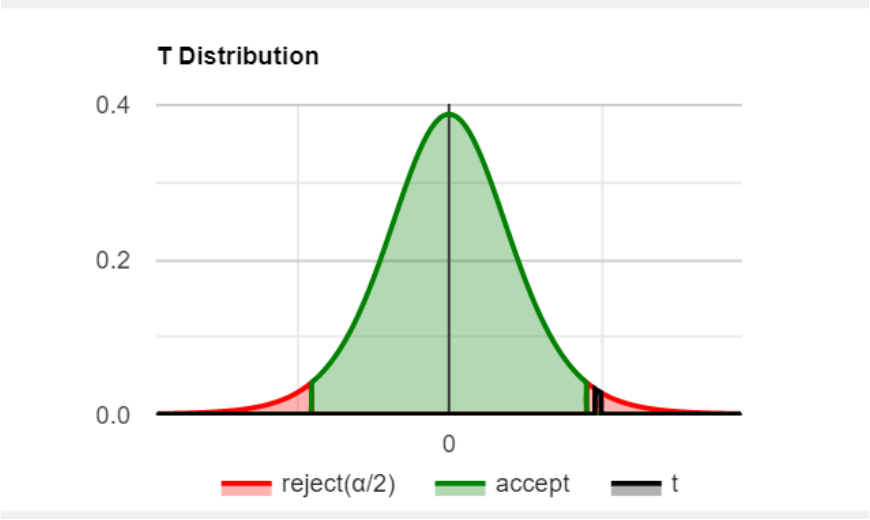

Figure 4.4.1. T Distribution of the paired t-test

The 95% confidence interval of this difference was found to be: From -3.08 to -0.12

Intermediate values used in calculations:

t = 2.4495 df = 9

standard error of difference = 0.653

	Basic AFL	**Modified AFL**
Mean	13.10	14.70
SD	8.02	9.60
SEM	2.54	3.04
N	10	10

Table 4.4.1. Paired t-test results for Basic AFL vs Modified AFL

4.5 Discussion

According to conventional criteria, a two-tailed P value of 0.0368 is considered to be statistically significant. This means that the null hypothesis can be dismissed.

The mean difference between the the number of bugs discovered was found to be 1.6 which means the modified AFL fuzzer on average detected more than 1 crash per binary more than the basic AFL fuzzer. This may not seem like a big difference but fuzzing generally is carried out by extremely powerful computers and for long periods of time. Due to time limitations, the study was only able to run the fuzzers for an hour making the mean difference of 1.6 quite significant.

The changes made to AFL significantly improved the performance as expected. Simply put, more bugs were found by the modified version in the same amount of time. This translated to the fact that the modified AFL runs faster than the basic version.

This decisively proves that adding the TCG IR to help translate the blocks does in fact significantly improve the performance of the AFL Fuzzer under QEMU mode in terms of speed and bug detection.

5. CONCLUSION

5.1 Introduction

While symbolic execution-based whitebox methodologies have been gaining prominence, their scalability is not comparable to that of blackbox or greybox fuzzers. Speeding up the greybox fuzzer AFL can help the entire SDLC and error free software can be published quickly.

The study was able to successfully answer the research question:

> "*Can implementing an intermediate layer in the tiny code generator of the American Fuzzy Lop fuzzer improve its speed significantly?"*

The answer simply put is yes. A more detailed discussion can be found in the next section.

5.2 Conclusion

In the experiment conducted, it was observed that the modified AFL fuzzer provided better performance in QEMU mode. In the same amount of time, the modified fuzzer was consistently able to detect more bugs. The mean difference between the bugs discovered was 1.6 which means the modified AFL was able to detect atleast one bug more in each of the binaries as compared to the basic AFL when run in QEMU mode.

This conclusively proves that adding the TCG IR to help translate the blocks does in fact significantly improve the performance in terms of speed and bug detection. The hypothesis was proved to be true. However, the study has certain limitations that need to be taken into consideration. These are discussed in the next section.

5.2 Limitations

1. Due to time constraints, each binary was tested only for one hour. This may give rise to discrepancies as the test can be carried out for weeks to discover all the bugs. It is essential to run the tests to completion and note the time required to discover all the known bugs in each binary. This will unequivocally prove whether the modified AFL fuzzer performs faster than basic AFL or not.

2. More binaries need to be tested to fully gauge the performance difference between the two fuzzers. It can be a co-incidence that the ten binaries selected gave favourable results.

3. Both fuzzers were run with chain caching enabled. The same experiment can be conducted without chain caching to be able to fully comprehend the difference in performance. The results may or may not be aligned with the findings of this study.

4. Only QEMU mode is affected by the modifications made. They have no effect on the basic usage of AFL. The basic AFL in normal mode will still give better results than the modified version.

5. Another factor that may affect dependability is the haphazardness in AFL's fluffing. Since AFL uses random inputs, the performance is different in each run. Several runs need to be conducted to ensure the consistency of the results.

6. The limitation here is that only one parameter, the number of bugs discovered, is being measured. However, there are more nuances to fuzzing that need to be taken into consideration. The time taken to discover all the bugs might be high but the fuzzer might be able to discover a majority of the bugs in a small amount of time and then take longer to discover the rest. This may be important in many applications where time is limited.

7. Another limitation is that the fuzzer might work excellently on the ten binaries being tested but might not work so well on other programs since each piece of software is unique in its structure and build. But these libraries have been selected because they are used as the benchmark by scientists (Cadar, Pawlowski, Dill & Engler, 2008) (Gligoric et al., 2010).

8. Only one device was used to conduct the experiment. The architecture differences on other hardware could create issues in the block chaining algorithm. It is important to test on devices with different architectures.

5.3 Further Research

Future research can include trying to combat the various challenges by combining the best methods discovered so far for each problem area. The following is a proposed implementation:

1. Before fuzzing commences, one can implement static program analysis by analysing the program control and data flow to produce an *input dictionary* using the strategies delineated by Shastry et al. (2017). This can aid the fuzzer in determining the seed inputs that will create the most number of paths thereby increasing code coverage.
2. The fuzzer can then be run for a limited time as a test run to allow the neural byte sieve to come up with optimal location for the next fuzzing cycle.
3. Then Driller's concolic execution engine can be implemented to split the program into compartments and create more paths using steps 1 & 2.
4. This can be done in conjunction with TaintScope's checksum verification algorithm to select the best mutated input seeds.
5. Lastly, AFLFast's algorithms can be used to trace rare paths.

Another avenue of research can include implementation of multiple fuzzers and running them simultaneously, sharing information with each other, speeding up the fuzzing process. As we have observed, contingent on the binary, different symbolic executors will be more or less effective and the right one can be chosen dynamically to allow for maximum code coverage while reducing duplication by marking paths that are traced.

REFERENCES

Arcuri, A., Iqbal, M., & Briand, L. (2012) Random testing: Theoretical results and practical implications. *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, (pp. 258-277).

Bastani, O., Sharma, R., Aiken, A., & Liang, P. (2017). Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, (pp. 95-110). ACM Press. https://doi.org/10.1145/3062341.3062349

Bayer, U., Moser, A., Kruegel, C., & Kirda, E. (2006). Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1), 67–77. https://doi.org/10.1007/s11416-006-0012-2

Böhme, M., & Paul S., (2016). A probabilistic analysis of the efficiency of automated software testing. *IEEE Transactions on Software Engineering*, vol. 42, no. 4, (pp. 345-360). https://doi.org/10.1109/TSE.2015.2487274

Böhme, M., Pham, V. T., & Roychoudhury, A. (2016). Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16* (pp. 1032-1043). Vienna, Austria: ACM Press. https://doi.org/10.1145/2976749.2978428

Bounimova, E., Godefroid, P., & Molnar, D. (2013). Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. *Proceedings of the 2013 International Conference on Software Engineering* (pp. 122-131). San Francisco, CA, USA: IEEE Press. Retrieved from https://dl.acm.org/citation.cfm?id=2486805

Burnim, J., & Sen, K. (2008). Heuristics for scalable dynamic test generation. *In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08*, (pp. 443–446). Washington, DC, USA. IEEE Computer Society. https://doi.org/10.1109/ase.2008.69

Cadar, C., Dunbar, D., & Engler D.R. (2008) KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, (pp. 209-224).

Cadar, C., Pawlowski, P.M., Dill, D. L., & Engler D. R. (2008). EXE: Automatically generating inputs of death. ACM Trans. *Inf. Syst. Security*, vol. 12, no. 2, (pp. 1-38).

Cha, S. K., Woo, M., & Brumley, D. (2015). Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy*, (pp. 725-741). IEEE. https://doi.org/10.1109/sp.2015.50

Dahse, J., & Holz, T. (2014). Simulation of Built-in PHP Features for Precise Static Code Analysis in NDSS '14, San Diego, CA, USA. Retrieved from https://www.syssec.ruhr-uni-

bochum.de/media/emma/veroeffentlichungen/2014/01/21/rips-NDSS14.pdf

Daniel, B., Dig, D., Garcia, K., & Marinov, D. (2007). Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* - ESEC-FSE '07, (pp. 158-162). ACM Press. https://doi.org/10.1145/1287624.1287651

DeMott, J., Enbody, R., & Punch, W.F. (2007) Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. In *BlackHat and DefCon 17'*. Retrieved From: https://www.blackhat.com/presentations/bh-usa-07/DeMott_Enbody_and_Punch/Whitepaper/bh-usa-07-demott_enbody_and_punch-WP.pdf

Dewey, K., Roesch, J., & Hardekopf, B. (2014). Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* - ASE '14, (pp. 725-730). ACM Press. https://doi.org/10.1145/2642937.2642963

Franz, M. (2010). A Maze of Twisty Passages all Alike: A Bottom-Up Exploration of Open Source Fuzzing Tools and Frameworks at *CERT Vulnerability Discovery Workshop*, February 2010

Fraser, G., & Arcuri, A., (2013). Whole test suite generation. IEEE *Transactions on Software Engineering*, vol. 39, no. 2, (pp. 276-291).

Gadi, E., et al. (2007). Open Source Fuzzing Tools. Syngress Publishing. "Chapter 2: Fuzzing –What's That?", (pp 11-26).

Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., & Marinov, D. (2010). Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* - ICSE '10 (pp. 225-234). ACM Press. https://doi.org/10.1145/1806799.1806835

Godefroid, P., Levin, M. Y., & Molnar, D. (2012). SAGE: Whitebox Fuzzing for Security Testing. *ACM Queue-Networks*, 10(1), (pp. 20-28). https://doi.org/10.1145/2090147

Havrikov, N. (2017). Efficient fuzz testing leveraging input, code, and execution. *In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion* (ICSE-C), (pp. 417-420). IEEE. https://doi.org/10.1109/icse-c.2017.26

Havrikov, N., Höschele, M., & Galeotti, A. (2014). XMLMate: Evolutionary XML test generation. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16, (pp.221-226). https://www.st.cs.unisaarland.de/testing/xmlmate/

Kargén, U., & Shahmehri, N. (2015). Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* - ESEC/FSE

2015. ACM Press. https://doi.org/10.1145/2786805.2786844
Leitner, A., Oriol, M., Zeller, A., Ciupa, I., & Meyer, B. (2007). Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* - ASE '07, (pp. 417-420). ACM Press. https://doi.org/10.1145/1321631.1321698

Moser, A., Kruegel, C., & Kirda, E. (2007). Exploring Multiple Execution Paths for Malware Analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)* (pp. 120-128). Berkeley, CA, USA: IEEE Press. https://doi.org/10.1109/sp.2007.17

Ognawala S., Hutzelmann, T., Psallida, E., & Pretschner, A. (2017). Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach. To be published In *Proceedings of the 33rd ACM/SIGAPP Symposium on Applied Computing (SAC)*, (pp. 80-89). Pau, France: ACM Press. https://arxiv.org/abs/1711.09362

Pak, B. (2012). Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *ACM Queue-Networks*, 12(2), (pp. 55-69). Retrieved From: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.635.3354&rep=rep1&type=pdf

Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., & Bos, H. (2017). VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, February 2017. Retrieved from https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., & Yang, X. (2012). Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12*, (pp 335–346). ACM Press. https://doi.org/10.1145/2254064.2254104

Shastry, B., Leutner, M., Fiebig, T., Thimmaraju, K., Yamaguchi, F., Rieck, K., Feldmann, A. (2017). Static Program Analysis as a Fuzzing Aid. In *Research in Attacks, Intrusions, and Defenses: 20th International Symposium*, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings (pp. 26–47). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-66332-6_2

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., … Vigna, G. (2016). SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In 2016 IEEE Symposium on Security and Privacy (SP). IEEE. https://doi.org/10.1109/sp.2016.17

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Christopher K., & Vigna, G. (2016). Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings 2016 Network and Distributed System Security Symposium*, (pp 101-112). San Diego, CA, USA: Internet Society. https://doi.org/10.14722/ndss.2016.23368

Takanen, A., Demott, J.D., & Miller, C. (2008). Fuzzing for software security testing and quality assurance. Artech House (pp 182-190).

Vimpari, M. An Evaluation of Free Fuzzing Tools. (2015) *Master's Thesis, University of Oulu*.

Vivanti, M., Mis, A., Gorla, A., & Fraser, G. (2013). Search-based data-flow test generation. *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, (pp. 370-379).

Wang, T., Wei, T., Gu, G., & Zou, W. (2010). TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *2010 IEEE Symposium on Security and Privacy* (pp. 497 – 512). Oakland, CA, USA: IEEE Press. https://doi.org/10.1109/sp.2010.37

Zeller, A., & Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), (pp. 183–200). https://doi.org/10.1109/32.988498

Non-peer-reviewed publications cited:

Biondo, A. Improving AFL's QEMU Mode Performance. *0x41414141*, 21 Sept. 2018, Accessed on 02 December 2018. Retrieved from abiondo.me/2018/09/21/improving-afl-qemu-mode/

Margaritelli, S. 2015. Fuzzing with AFL-Fuzz, a Practical Example (AFL vs Binutils). Accessed on 25 October 2018. Retrieved from https://www.evilsocket.net/2015/04/30/fuzzing-with-afl-fuzz-a-practical-exampleafl-vs-binutils/

Sigma. (2009). Statistical Analysis 3: Paired t-test. Sigma Center for Excellence in Mathematics & Statistics Support.

Zalewski, M. American Fuzzy Lop Accessed on 10 September. Retrieved from http://lcamtuf.coredump.cx/afl/technical_details.txt